ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4

Tomohiro Oda
Software Research Associates, Inc.
tomohiro@sra.co.jp

Keijiro Araki
Kyushu University
araki@csce.kyushu-u.ac.jp

Peter Gorm Larsen
Aarhus University
pgl@eng.au.dk

Abstract

It is possible to make Integrated Development Environments supporting formal methods that can be as flexible as the support for dynamic programming languages. This paper contributes with a demonstration employing different support environments for the Vienna Development Method Specification Language (VDM-SL) and design by contract for visual programming language. This includes ViennaTalk developed on top of Pharo 4 providing Smalltalk-styled LIVE browsers, VDM-SL interpreters, Smalltalk code generators, UI prototyping environments and a prototype Web API server to enable rigorous and flexible modeling during exploratory phases of software development. ViennaTalk uses the Slot mechanism in Pharo to test invariant assertions on instance variables in Smalltalk objects generated from VDM-SL specifications. In addition, we present a plugin named Assertch for Phratch, a scratch-clone visual programming environment on top of Pharo 4, that provides assertion blocks for designing and debugging a series of blocks.

Both ViennaTalk and Assertch combine flexible live modeling or coding while still supporting rigorous checking. ViennaTalk has been evaluated by experienced professional engineers of VDM-SL while Assertch has been evaluated by undergraduate students of computer science. ViennaTalk and Assertch both demonstrate that Pharo and its contemporary features support rigorous modeling in formal specification languages as well as flexible prototyping in Smalltalk.

Keywords Live environment, Lightweight formal methods, Specification animation, Validation

1. Introduction

Capturing user requirements and validating them by exploratory modeling techniques such as prototyping is a very successful method in software development. Smalltalk is a leading environment for quickly prototyping in a live fashion.

The lightweight use of formal methods has been proposed quite some time ago as a cost-effective way to promote industrial usage [1, 4]. The Vienna Development Method (VDM) [3] contains different specification language dialects known to support lightweight formal methods. In this paper we will concentrate on the VDM Specification Language (VDM-SL) which is an ISO standard [6]. VDM-SL is a formal specification language that has an executable subset, yet provides strong abstraction by powerful constructs with mathematical foundations. Many successful cases of practical application of VDM to large scale software systems are reported [14].

Existing tools for VDM-SL include type checkers, proof obligation generators, interpreters and code generators. IDEs such as VDMTools [4, 5] and Overture tool [7] integrate such tools, and are used in industrial software development, including but not limited to high assurance systems. The formal specification phase is not only to mathematically prove safety properties of the specification, but also to learn the target domain and elicit implicit requirements from clients and domain experts. A process of trial and error is often carried out in the early stages of the specification phase.

We developed ViennaTalk, a formal specification environment on top of Pharo 4, to introduce Pharo's flexible and live way of modeling. We also implemented Assertch, a plug-in for the visual programming environment Phratch[1], that introduces assertion blocks. In this paper, an overview of ViennaTalk and Assertch is provided in Sections 2 and 3. Components of ViennaTalk, such as browsers, interpreters, code generators, UI prototyping environments, and a prototype web API server, are explained and discussed.

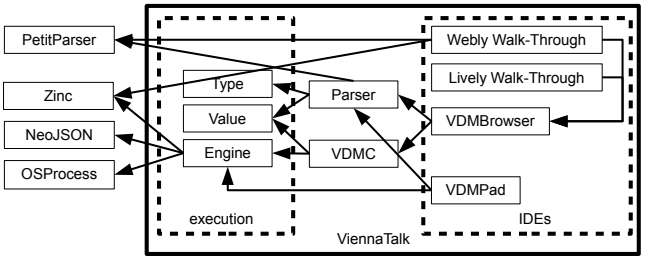

Figure 1. Packages and dependencies in ViennaTalk

2. ViennaTalk

ViennaTalk is a formal modeling environment for VDM-SL built upon Pharo 4.

Pharo is an Integrated Development Environment (IDE) where developers of the IDE and application developers use the same environment. Modification to Pharo is open to any application programmer. ViennaTalk shares the same design rationale. ViennaTalk is not a standalone IDE but serves as a part of the Pharo environment.

[1] http://www.phratch.com/

IWST '16 August 23, 2016, Prague, Czech Republic

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

ViennaTalk consists of the execution, IDEs and other utility packages. Figure 1 shows ViennaTalk's packages and their dependencies. The Engine package provides interpreters implemented as wrappers to external interpreters. The Type and Value packages provide types and values of the VDM-SL language as Smalltalk objects. The VDMC package manages states of animation sessions and translation between VDM-SL values and Smalltalk objects. ViennaTalk also has the Parser package including abstract syntax tree, pretty printers, a syntax highlighter and code generators.

ViennaTalk provides four development environments on top of the Pharo environment. The core also includes the VDM Browser as a UI to edit and evaluate the source code of VDM-SL. Developers can extend or implement a tool specialised to a particular development project. The full configuration of ViennaTalk includes tools such as VDMPad, Lively Walk-Through and Webly Walk-Through.

In this section, a brief overview of VDM-SL is presented. Browsers for VDM-SL, animation mechanisms, and prototyping environments are described afterwards.

2.1 Overview of VDM-SL

VDM-SL is a model-oriented formal notation based on discrete mathematics and logic that supports the description of both data and functionality. Although VDM-SL is not a programming language, it has a subset so that interpreters can simulate execution of the specified system [8]; simulated execution is called *animation*. An animation of the specification consists of a series of evaluations of expressions. In this paper, we only use the executable subset of VDM-SL.

A specification in VDM-SL has one or more modules. Each module defines types, constant values, functions, a state space and operations. Data are defined by means of types built using constructors that define structured data with records and tuples and collections such as sets, sequences and mappings from basic values such as Booleans and numbers. Constants and functions are referentially transparent, and operations may or may not read and/or change values of state variables defined in the state space. Each function or operation may have a precondition and a postcondition. A precondition constrains the state and arguments before its evaluation, and a postcondition constrains the state and arguments after the evaluation. The state space in each module may have an invariant that the state should always satisfy. A type may also have invariant that all values of the type should satisfy. If evaluation of an expression violates either invariants, preconditions or postconditions, the interpreter reports the violation and stops the animation.

VDM-SL has a powerful pattern matching mechanism including set unions, sequence concatenation and map unions. For example, a pattern `header ^ [1, 2, 3] ^ trailer` matches to `[0, 1, 2, 3, 4, 5]` with binding that `header` = `[0]` and `trailer` = `[4, 5]` where the ^ operator is sequence concatenation. VDM-SL also provides rich and powerful operators on maps, such as composition and restriction to the domain and the range of a map. Those features are useful for mathematical modeling. Although those features are executable, the objective of VDM-SL specification is to describe properties that the target system should satisfy.

2.2 Browsers in ViennaTalk

ViennaTalk provides two live browsers for VDM-SL inspired by the Smalltalk environment. One is a web-based IDE named *VDMPad*, the other is *VDM Browser* on Pharo's desktop. VDMPad is designed to be easy to use with small scale specifications that can be expressed in a single module. The VDM Browser targets small to medium scale specifications with multiple modules.

Both browsers are live; when a specification is executed, the user can modify the specification and the animation continues with the modified specification. Conventional animation tools do not allow the user to modify the values of the state variables. An animation always starts with the initial state, and modification to the specification forces the animation to restart with the initial state. ViennaTalk gives more freedom to the user; the user can change the values of the state variables, can modify the specification, and continue the animation by evaluating the next expression. If the state violates the specification by evaluation of an expression, the state before the evaluation will be kept. If the state violates the specification by modification of the specification, the state will be initialised.

2.2.1 VDMPad

VDMPad [9, 10] is a simple Web IDE for the formal specification language VDM-SL designed to support exploratory stages of modeling. A VDMPad server provides a VDM animation service to web clients for VDM-SL[2]. It is designed to supply lightweight, easy-to-use functionality comparable to a calculator in an engineer's pocket. VDMPad has two areas of intended use: formal methods education and exploratory development. VDMPad can also be used as a VDM-SL interpreter server for ViennaTalk installed on remote hosts.

One benefit of programming on a web browser is convenience. Downloading, installing and updating an IDE, along with libraries that the IDE depends on, can be a burden for beginners. VDMPad does not require any special plugin on the client side. Having a VDMServer running on the Internet, a prospective VDM learner can give it a try just by opening a browser and entering the URL at a classroom or a development site. Because the VDM interpreter runs on the server side, the client side is lightweight.

All operations to animate a VDM-SL specification on VDMPad are performed via the one, simple page. Figure 2 shows the major UI components of VDMPad. The user interface consists of five parts: a specification editor, an area for information about the state, a workspace, a return value display, and finally a message area. In addition, a retractable menu pane is provided.

The specification editor has a syntax highlighting feature to help the user see the structure of the specification. The user does not have to save the specification to animate it. Evaluating an expression in the workspace will send the specification to the server to animate it, and the specification will be saved on the local storage in the browser automatically along with the content of the workspace and other information about the status of the animation.

VDMPad displays the value of each of the state variables in the state area. The value of each variable is printed in an input field so that the user can edit the value. VDMPad can optionally display a diagram representation of the value of each state variable for intuitive understanding of the structure of the value.

The workspace is a text editor named after its counterpart in the Smalltalk environment. The main purpose of the workspace is to enter and organise a collection of expressions for evaluation, but the user can also write notes inside this area. The user can evaluate such an expression by selecting it on the workspace and clicking the `evaluate` button (See Figure 2). The return value of the expression is displayed below the `evaluate` button. The return value can also be displayed in a diagram representation. If evaluation generates a run-time error, the error message is displayed in the message area at the bottom of the window.

2.2.2 VDM Browser

Figure 3 shows VDM Browser. Like Smalltalk's class browsers, VDM Browser styles syntax by colours and also pretty prints the

[2] A similar web-based IDE can be found in [12].

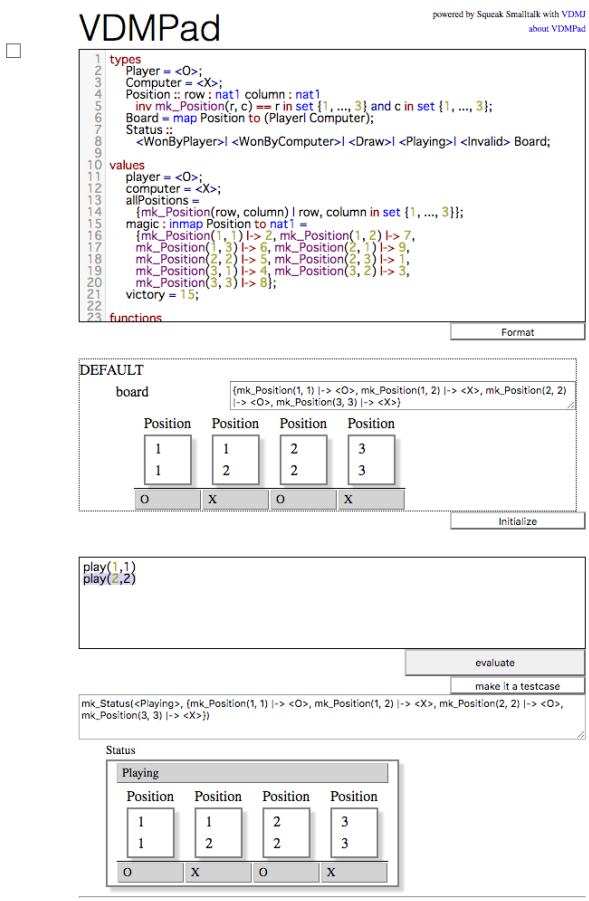

Figure 2. A screenshot of VDMPad

source either automatically or by an explicit command in the context menu.

The VDM Browser has two roles; a source editor and an inspector on an animation context of the specification. The upper left pane is the module list where the modules defined in the specification are listed. By selecting a module in the module list, the user can choose the module to edit the source and to evaluate an expression. The upper middle pane is the variable list where the state variables of the selected module are displayed (or for the entire system if no module is selected). The upper right pane is the value pane where the value of the selected state variable is displayed. The user can also enter an expression in the value pane and assign it to the state variable.

VDM Browser has a workspace. The workspace appears by selecting the Workspace tab. Figure 4 shows a workspace after evaluating the expression `getGoal()`. The return value of the evaluated expression is inserted and selected (`9000`). An expression is evaluated in the scope of the selected module in the module list.

2.3 Animation Mechanisms

2.3.1 Interpreters

ViennaTalk uses VDMJ, a VDM interpreter in Java, as an external interpreter [2]. ViennaTalk provides three kinds of wrappers to use VDM interpreters: ViennaVDMJ, ViennaClient and Vienna-

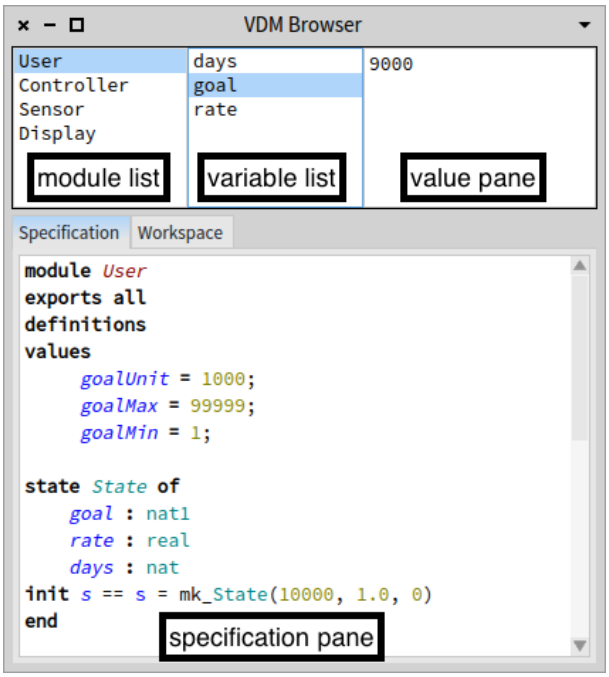

Figure 3. A screenshot of the VDM Browser

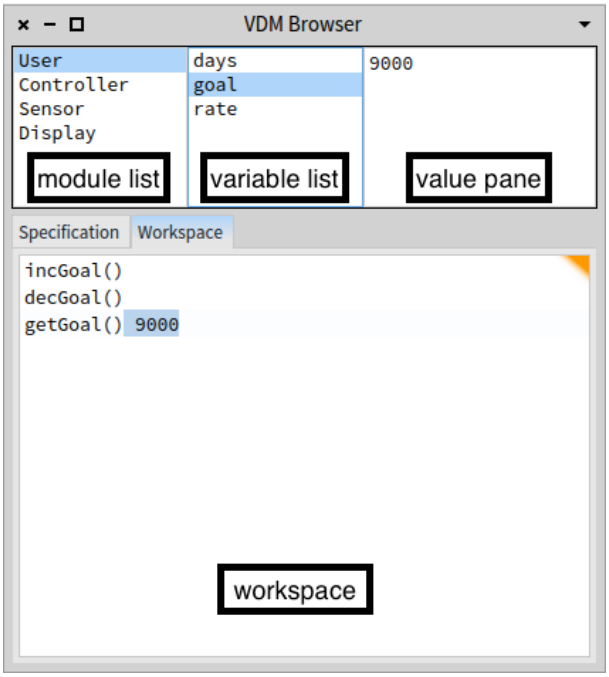

Figure 4. A screenshot of Workspace in VDMBrowser

BankEngine. ViennaVDMJ uses VDMJ[3] to evaluate an expression via OSProcess[4]. ViennaClient is a wrapper that uses VDMPad/ViennaTalk server via HTTP. ViennaBankEngine is a pool of wrappers to distribute computational loads over the wrapper pool. All

[3] https://github.com/nickbattle/vdmj

[4] http://www.squeaksource.com/OSProcess.html

```
operations
    sieve : nat1 ==> ()
    sieve(x) ==
        space := [space(i)
                    | i in set inds space
                   & space(i) mod x <> 0];
```

Figure 5. A part of Sieve of Eratosthenes in VDM-SL

```
sieve: x
    space := ((1 to: space size)
        select: [ :i | (space value: i) \\ x ~= 0 ]
        thenCollect: [ :i | space value: i ])
        asOrderedCollection
```

Figure 6. The sieve Smalltalk method generated from the VDM-SL specification in Figure 5

those wrappers respond to the #evaluate:specification:states: message.

The wrappers are sessionless, which means that a wrapper does not store any state information about an animation; a wrapper can be used by multiple animations and an animation can be carried out by different wrappers for each evaluation of an expression.

2.3.2 Code Generators

ViennaTalk also has three kinds of Smalltalk code generators. ViennaVDM2Smalltalk generates a string that can be evaluated by Smalltalk compilers. ViennaVDM2SmalltalkClass generates a class for each module in the VDM-SL specification. ViennaVDM2SmalltalkObject generates an anonymous class for each module in the VDM-SL specification and creates its instance.

The generated code uses standard Smalltalk classes if possible. For example, characters in VDM-SL are translated into objects of the Character class in the standard class library. A set {`1, 2, 3, 4`} in a VDM-SL specification is translated into {`1. 2. 3. 4`} `asSet`.

One goal of the ViennaTalk's code generators is to output human-readable and natural Smalltalk programs. The following specification is a snippet from an executable specification of Sieve of Eratosthenes to compute a series of prime numbers.

ViennaTalk defines classes for values and types of VDM-SL. VDM-SL and Smalltalk have different type systems. VDM-SL is statically typed, and Smalltalk is dynamically typed. Smalltalk has multiple concrete classes for integers: `SmallInteger`, `LargePositiveInteger`, and `LargeNegativeInteger` in Pharo 4. VDM-SL also has multiple types for integer and its subtypes: **`int`**, **`nat`**, and **`nat1`**. It is not straightforward to define a mapping between Smalltalk classes and VDM-SL types.

ViennaTalk provides classes to express VDM-SL types independent of the classes of instance objects translated from VDM-SL values. VDM-SL types are instances of subclasses of the ViennaType class and VDM-SL values are instances of standard classes such as Integer, Float, Array and so on. The instance objects translated from VDM-SL values are loosely associated with VDM-SL type objects by the `#includes:` message. An instance of the ViennaType class responds to the `#includes:` message and answers whether the object given as the argument belongs to the receiver's type. For example, a SmallInteger object `1` is translated from VDM-SL's value `1`, and the expression `ViennaType nat includes: 1` evaluates to `true`.

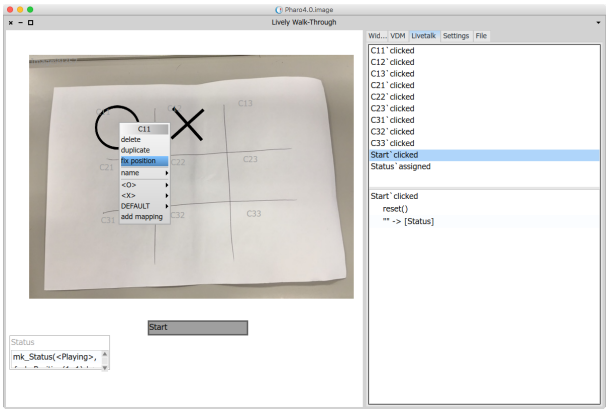

Figure 7. A screenshot of Lively Walk-Through

Those type objects also respond to the >= message to answer the subtype relationship in VDM-SL's type system. Table 1 shows the mapping between VDM-SL types and their type objects and value objects in Smalltalk. Those classes for VDM-SL types and values are not only for code generators but they can be used to implement other tools to manipulate VDM-SL specifications.

ViennaTalk uses the Slot mechanism [13] to implement invariant checking on instance variables. The Slot mechanism enables application programmers to define models of instance variables without customizing the compiler. A Slot defines a series of bytecodes that the compiler should generate for read access to the variables, and another series for write access. A class can be defined optionally with pairs of an instance variable name and a slot. When the compiler generates bytecodes of read or write access to an instance variable in a mehtod, the compiler delegates the bytecode generation task to the corresponding slot. A slot may also have methods to process read and/or write access to an instance variable of an object in addition to bytecodes.

ViennaTalk provides ViennaStateSlot that sends a `#inv` message to an object when an instance variable of the object is assigned a value. The `#inv` method tests the invariant. If the invariant evaluates to `false`, ViennaStateInvariantViolation will be signalled. The code generator generates the `#inv` method from the state space definition and uses ViennaStateSlot for each state variable when the code generator defines a class for a VDM-SL module. ViennaStateSlot enables the state invariant to be checked in the Smalltalk system, and helps Smalltalk programmers to detect an unexpected object assigned to an instance variable.

2.4 Prototyping Environments

ViennaTalk includes two prototyping environments, Lively Walk-Through [11] and Webly Walk-Through [11].

Lively Walk-Through is a UI prototyping environment that combines a formal specification in VDM-SL and prototypical UI design. Stakeholders can use the prototype to check whether the specification in VDM-SL satisfies the requirements by operating the specified system through the user interface. Figure 7 shows a prototype of the Tic-Tac-Toe game on Lively Walk-Through. The left part is the prototypical user interface. In Figure 7, an image of 3x3 board, two stone widgets (O and X), a start button, and a status display are placed. The right part has tabs of a widget catalog, a VDM Browser, a LiveTalk browser where the UI events are associated with VDM expressions and data flow to widgets, settings menu, and file menu. The user can modify the specification and the state using VDM Browser when operating on the prototypical user interface.

Type description	VDM type	Smalltalk expression	Implementation class	Value's class
natural number	nat	ViennaType nat	ViennaNatType	Integer
non-zero natural number	nat1	ViennaType nat1	ViennaNat1Type	Integer
integer	int	ViennaType int	ViennaIntType	Integer
real number	real	ViennaType real	ViennaRealType	Float
boolean	bool	ViennaType bool	ViennaBoolType	Boolean
quote	<quote>	ViennaType quote: #*quote*	ViennaQuoteType	Symbol
option type	[*t*]	*t* optional	ViennaOptionType	*t*'s class or UndefinedObject
product type	*t1* * *t2*	*t1* * *t2*	ViennaProductType	Array
union type	*t1* | *t2*	*t1* | *t2*	ViennaUnionType	*t1* or *t2*'s class
set type	set of *t*	*t* set	ViennaSetType	Set
seq type	seq of *t*	*t* seq	ViennaSeqType	OrderedCollection
non-empty seq type	seq1 of *t*	*t* seq1	ViennaSeq1Type	OrderedCollection
map type	map *t1* to *t2*	*t1* mapTo: *t2*	ViennaMapType	Dictionary
injective map type	inmap *t1* to *t2*	*t1* inmapTo: *t2*	ViennaInmapType	Dictionary
partial function type	*t1* -> *t2*	*t1* -> *t2*	ViennaPartialFunctionType	BlockClosure
total function type	*t1* +> *t2*	*t1* +> *t2*	ViennaTotalFunctionType	BlockClosure
composite(record) type	compose *t* of *f1* : *t1* *f2* : *t2* *t3* end	ViennaType compose: '*t*' of: {*f1* . false . *t1*. {*f2* . true . *t2*}. {nil . false . *t3*}}	ViennaCompositeType	ViennaComposite
type invariant	*t* inv *pattern* ==*expr*	*t* inv: [:*v* | *expr*]	ViennaConstrainedType	*t*'s class

Table 1. VDM-SL types and corresponding Smalltalk expressions, type classes and value classes

Webly Walk-Through is a web API server that publishes operations defined in a VDM-SL specification as web APIs. Used in conjunction with HTML and JavaScript, Webly Walk-Through can be used as a prototyping tool for web applications. Web APIs on Webly Walk-Through exchange data in JSON format.

2.5 Discussion and Related Work

We have built a formal methods environment named ViennaTalk on top of Pharo 4. The development of ViennaTalk demonstrated that a live environment as an extremely dynamic environment benefits development environments for languages oriented to static analysis.

VDMTools [4, 5] and the Overture tool [7] are two major IDEs for VDM family practically used in industries. Both VDMTools and Overture tool aims at rigorous large scaled development; they provide syntax and type checking, interpreters and code generators for large specifications modularised and stored in directory trees. Interpreters provided by VDMTools and Overture tool are not live; an animation session restart with the initial state when the specification is modified. ViennaTalk's live animation makes the initial stage of the modeling process more flexible and more efficient by allowing animation and modification interleaved with each other.

The code generators of VDMTools and Overture require external development tools for the target language such as compilers and linkers. ViennaTalk can generate Smalltalk objects directly from a VDM-SL source and run it instantly. ViennaTalk can also generate a class library. Having a class library generated from a VDM-SL source, the user can browse classes and methods to gradually and seamlessly integrate the code with other components of the Pharo environment in the live way.

A prototype WebIDE for Overture tool has been developed [12]. ViennaTalk provides a web-based IDE named VDMPad as described in Section 2.2.1. Two web IDEs share the same strengths, in both industrial and educational settings, that the tools can be delivered and updated by the central controlling manner. They have different orientation inherited from their base environment: the Eclipse-based Overture tool and the Smalltalk-based ViennaTalk. Overture's WebIDE has a file-based user interface based on the Eclipse-based standalone Overture tool, and provides functionality using components of the Overture tool. VDMPad has an image-based user interface which does not employ the concept of files except importing/exporting snapshots from/to local file systems. Both ViennaTalk and VDMPad are designed as personal modeling environments for exploratory modelling tasks while Overture and its WebIDE pursuit rigorous modeling in either personal or team-based developments.

ViennaTalk does not only use Pharo's class libraries, but also adopted liveness and UI design inspired by Pharo. We received positive feedback from the experts. They commented that the workspace is convenient in trial-and-error to understand the system's behaviour. We also asked expert VDM-SL engineers to use Lively Walk-Through in hackathon sessions. They commented that the UI prototyping is a good tool to capture the requirements. We believe the liveness and the exploratory modeling approach supported by ViennaTalk can be accepted by formal engineers. We have been providing a public VDMPad server at: `http://vdmpad.csce.kyushu-u.ac.jp` and invited expert VDM-SL engineers from industry for evaluation.

Smalltalk programmers can also enjoy benefits of lightweight formal methods. The invariant checking mechanism implemented in ViennaTalk can help Smalltalk programmers. Use of the Slot mechanism like ViennaStateSlot gives confidence that an object maintains data integrity. ViennaStateSlot sends a `#inv` message when an instance variable is overwritten. The `#inv` method checks the sanity of the receiver object. The `#inv` method can serve as debug code as well as design documentation that describes the assumed constraints of the object.

3. Assertch

Assertch is a plug-in extension to Phratch[5], a live and visual programming environment on Pharo 4.

The objective of Assertch is to introduce formal methods to undergraduate computer science students. Assertch provides four

[5] http://www.phratch.com/

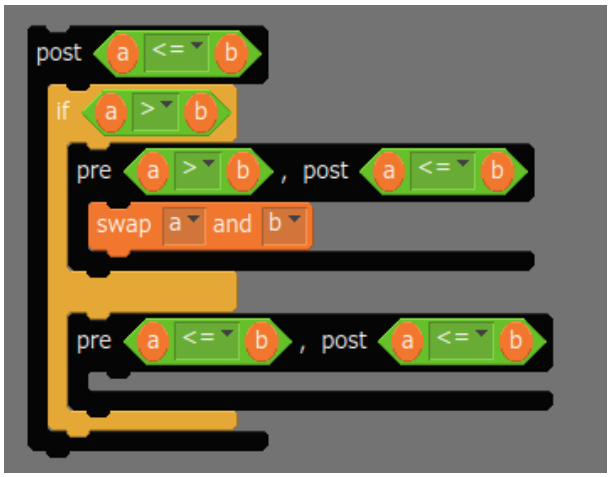

Figure 8. An example program using Assertch's assertion blocks

kinds of assertion blocks, an error reset block and operator blocks for convenience to write assertions. Using assertion blocks, students learn design by contracts including the concept of assertions, how to specify a code block, how to implement the code block to satisfy the specification, how to explain the implemented code block using assertions, how to debug the code using assertions and how to comprehend the code using assertions.

Figure 8 is an example program using assertion blocks. The black blocks are assertion blocks. The outer black block is a postcondition block, and the two inner black blocks assert both precondition and postcondition. Those assertion blocks are used for runtime checking and as design documentation.

For example, we can explain why the program in Figure 8 works. As the outmost postcondition block says, the goal of the program is to make `a` less than or equal to `b`. If `a` is greater than `b`, we have a precondition `a > b` and a postcondition `a <= b`. Swapping `a` and `b` will obviously satisfy the postcondition. If `a` is not greater than `b`, we have a precondition `a <= b` and a postcondition `a <= b`. Because the precondition and the postcondition are identical, nothing is needed. Therefore, in either case the program satisfies the postcondition `a <= b`.

3.1 Evaluation

Three undergraduate students majoring computer science used Assertch for simple programming tasks. All of the three students were new to Phratch or Scratch-based visual programming environments. The students received a 10 minute brief lecture on how to use the visual programming environment and assertion blocks. The students performed two programming tasks using assertion blocks and then they were interviewed individually. In the interview, students explained their programs using preconditions and postconditions. All of them responded that they understood the functionality of assertion blocks and that they consider assertion blocks useful to correctly construct a program and to explain the program. They also answered that they consider assertions useful in textual programming languages.

4. Concluding Remarks

ViennaTalk and Assertch are built upon the Pharo environment. We believe that building lightweight formal tools upon dynamic environment can be of mutual benefit. Smalltalk's flexibility and liveness help exploratory formal specification, and partial use of formal specification in Smalltalk development gains confidence in the model for little slow-down of Smalltalk's rapid programming. One significant aspect is Slot. Slot is one of the contemporary features in Pharo that allows the flexibility to change semantics of instance variables. ViennaTalk uses the Slot mechanism to implement state invariants on instance variables. It demonstrates that Pharo's flexibility helps implementation of lightweight formal tools that aid rigorous construction of program code. We continue developing ViennaTalk to pursue rigorous and flexible modeling for efficient and reliable software development.

Acknowledgments

This work is supported by Grant-in-Aid for Scientific Research (S) 24220001 and Grant-in-Aid for Scientific Research (C) 26330099. We would also like to thank Nick Battle and Paul Chisholm for valuable feedback on drafts of this paper.

References

[1] S. Agerholm and P. G. Larsen. A Lightweight Approach to Formal Methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, Boppard, Germany, October 1998. Springer-Verlag.

[2] N. Battle. VDMJ User Guide. Technical report, Fujitsu Services Ltd., UK, 2009.

[3] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. doi: 10.1145/1668862.1668879. ISBN 0-521-62348-0.

[4] J. Fitzgerald, P. G. Larsen, and S. Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2): 3–11, February 2008. doi: 10.1145/1361213.1361214.

[5] P. G. Larsen. Ten Years of Historical Development: "Bootstrapping" VDMTools. *Journal of Universal Computer Science*, 7(8):692–709, 2001.

[6] P. G. Larsen, B. S. Hansen, et al. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996. International Standard ISO/IEC 13817-1.

[7] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010. ISSN 0163-5948. doi: 10.1145/1668862.1668864. URL `http://doi.acm.org/10.1145/1668862.1668864`.

[8] K. Lausdahl, P. G. Larsen, and N. Battle. A Deterministic Interpreter Simulating A Distributed real time system using VDM. In S. Qin and Z. Qiu, editors, *Proceedings of the 13th international conference on Formal methods and software engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 179–194, Berlin, Heidelberg, October 2011. Springer-Verlag. doi: 10.1007/978-3-642-24559-6_14. URL `http://dl.acm.org/citation.cfm?id=2075089.2075107`. ISBN 978-3-642-24558-9.

[9] T. Oda and K. Araki. Overview of VDMPad: An Interactive Tool for Formal Specification with VDM. In *International Conference on Advanced Software Engineering and Information Systems (ICASEIS) 2013*, Nov 2013.

[10] T. Oda, K. Araki, and P. G. Larsen. VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In N. Plat and S. Gnesi, editors, *FormaliSE 2015*, pages 33–39, Florence, May 2015. In connection with ICSE 2015.

[11] T. Oda, Y. Yamomoto, K. Nakakoji, K. Araki, and P. G. Larsen. VDM Animation for a Wider Range of Stakeholders. In F. Ishikawa and P. G. Larsen, editors, *Proceedings of the 13th Overture Workshop*, pages 18–32, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan, June 2015. Center for Global Research in Advanced Software Science and Engineering. URL `http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf`. GRACE-TR-2015-06.

[12] R. S. Reimer and K. D. Saaby. An Open-Source Web IDE for VDM-SL. Master's thesis, Department of Engineering, Aarhus University, Denmark, May 2016.

[13] T. Verwaest, C. Bruni, M. Lungu, and O. Nierstrasz. Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 959–972, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048138. URL http://doi.acm.org/10.1145/2048066.2048138.

[14] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4): 1–36, October 2009. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1592434.1592436.